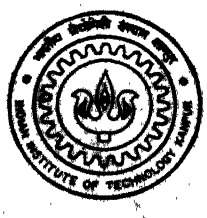
DESIGN AND IMPLEMENTATION OF A WORKFLOW MODEL

Part I : Task Model

By

K. SRINIVAS



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPU

JANUARY, 1997

Design and Implementation of a workflow model

Part I : Task Model

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

K. Srinivas

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

Jan. 1997

CERTIFICATE

This is to certify that the work contained in the thesis titled Design and Implementation of a Workflow Model, Part I : Task Model by K. Srinivas has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dr. Tapas K. Nayak,
Assistant Professor,
Dept. of CSE,
IIT Kanpur.

Dr. Ratan K. Ghosh,
Associate Professor,
Dept. of CSE,
IIT Kanpur.

CSE-1997-M-SRI-DES

Acknowledgements

I express my sincere gratitude to my supervisors Dr. Tapas Nayak and Dr. R.K. Ghosh for their constant support throughout the thesis. I am deeply indebted to Dr. Tapas Nayak for his excellent guidance and also personally. He was a source of inspiration at work. I thank Dr. Ghosh for spending his valuable time for me.

This thesis would have been incomplete, but for the companionship of Kaladhar. We often competed for postponing the deadlines.

I would like to thank all my friends of mtech95, who made my life enjoyable at IITK. Thanks to all my colleagues at the CSE lab, at which I spent most of my time. Thanks to Brahmaji for his cooperation. I personally thank prav, rk, panku, sameer, dsri, raghu, gore and all my friends who made Hall IV a great place to stay.

My special thanks go to the great *csealpha3*, which is host to every line of code of this thesis.

I dedicate this thesis to the members of my family, without whose moral support and love, I could not have come this far.

Abstract

Workflows *are activities involving co-ordinated execution of multiple related* tasks. *The control flow and data flow among these tasks are defined by a* Workflow Model. *This thesis proposes a workflow model which allows for passing data between tasks by means of* objects *and provides features for defining and monitoring a workflow. A prototype implementation of the model has been carried out with* **Ingres** *as the database management system. To illustrate the concepts, a bank information system has been developed as an example.*

This work has been divided into two parts : the task model *and* object model. *This thesis covers the task model; the object model is presented in [10].*

The task model involves the management and co-ordination of the tasks in the workflow. Control flow among tasks can be specified by means of event-condition-action *rules. Specification languages have been designed for describing the workflow and defining the objects. The task model specifies the methods of interaction between various tasks and routing of objects between them. The* Task Server *ensures these specifications by task scheduling and event scheduling.*

Contents

CHAPTER 1

Introduction

Many enterprises use multiple information processing systems that were developed independently to automate different functions. These systems require the participation of multiple applications and databases. Examples of such applications include service order processing, travel reservation and transactions in a bank. Transaction-oriented information systems provide only limited support for managing the control and data flow in such applications. A workflow model uses a declarative control and data flow specification language to facilitate a separation between the application code and the control and data flows between different tasks of the application.

1.1 What is workflow?

Workflows are activities involving coordinated execution of multiple *tasks* performed by different *processing entities* [11]. These activities tend to be of long duration, involve many users and tools over a heterogeneous and distributed environment. A *task* defines some work to be done and can be specified in a number of ways. A *processing entity* performs the tasks; it may be a person, or a software system (e.g., a mailer or an application program). The execution of the tasks is controlled by a *Workflow management system* (WFMS).

A task is the basic unit of activity in a workflow. The task is typically an application program to do some well-defined processing of objects e.g., the verification of

the validity of a cheque in a bank application. The task may be a fully automated one, or an interactive application which requires user attention. For example, the signature verification task may require constant interaction with the user, whereas the account updating task may run in the background.

Corresponding to a typical work environment, the tasks may be **distributed** across a network of machines. Each task may be run by a particular user on a machine. It is natural to assign roles to the users and classify the tasks by specifying who can execute a task and in what roles.

Every user (or task) has, at any time, a *worklist* – a queue of *job*s waiting to be processed by this user (or task). A *job* corresponds to the unit of work performed by a task e.g., a cheque in a bank application. A job, after getting its processing done at a task, goes to the next task in the pipeline. The rules for passing the objects may be quite complex, depending on the application. For instance, some objects get divided at certain tasks and are routed through different tasks. They maintain independent existence until the required processing is done and merge at some other task. It is possible to generate copies of objects and route them independently. This is how essentially work objects *flow* through the system.

1.2 History and Related Work

With the increasing prevalence of local area networks, the necessity for software which allows for group collaboration is being felt. Such software is termed *groupware.* Lotus Notes is an example. They provide capabilities such as a sophisticated messaging system, document processing, database support, etc. However, groupware has some serious limitations : there is no notion of a *transaction* and the associated concepts of locking and logging. Most groupware products do not also have a higher-level process definition capability. Perhaps, it may be appropriate to say groupware provides only *ad hoc* workflow capabilities.

Full-fledged Workflow models are designed to address these problems. They are based on the concepts of extended transaction models. They provide ways of defining the *processes* (i.e., activities). They provide for failure recovery and maintaining the

semantics of transactions.

There are many transaction models that have been proposed to handle multi-database applications. These cannot be adopted for designing and implementing workflows due to the following reasons :

1. The extended transaction models have a pre-defined set of properties that may or may not be required by the semantics of a particular activity.

2. The processing entities in a workflow may not provide support for facilities implied by a transaction model.

The motivation for the development of workflow models was derived from the desire to overcome the limitations of groupware as well as transaction models. Workflow models are, in fact, seen as the future of advanced transaction models [2]. Some issues related to workflows are addressed in the work on long running activities [3, 4].

Some of the early workflow products that have been developed were email-based. The jobs are routed through email. But this is rudimentary and not very flexible. Of late, workflow systems find applicability on the *World Wide Web*. The web provides a networking backbone, which the tasks may use conveniently. Web-based workflow systems have several other advantages. They allow tasks to spread wide across the Internet. A database located at any site can be used easily by any other machine on the Internet. See [1] for an example of such a system.

A number of commercial and research workflow systems are in development, like the **Exotica** system at the IBM's **Almaden Research Center** [9] and **FileNet**'s commercial product [7]. The **Workflow Management Coalition** [14] is developing a set of standards for workflow software to increase the interoperability and connectivity between different workflow products.

1.3 Organization of the thesis

This thesis deals with the task management part of a workflow. The object management part of it is discussed in [10]. These two theses together cover the whole

of design and implementation of a workflow system. These two theses are closely related; so there are implicit references to [10] throughout this thesis.

The rest of the thesis is organized as follows :

Chapter 2 describes the basic concepts of a workflow model and a workflow management system. It presents a conceptual overview of a workflow system.

Chapter 3 presents a functional overview of a workflow model by describing its components and their functionality. The proposed workflow model is described here in addition to the WfMC reference model.

The next chapter discusses in more detail, the design and implementation of the *task model*, which deals with the tasks and their management.

The *task server*, which monitors the tasks and provides other services is discussed in detail in chapter 5.

We conclude the thesis with chapter 6 and discuss possible extensions to this work.

CHAPTER 2

Workflow Concepts

In this chapter, we present the concepts behind a workflow model, and how it relates to the advanced transaction models. Then we discuss an example to see these concepts in application. To appreciate the significance of workflow models, we first discuss the drawbacks of existing transaction models.

2.1 Advanced Transaction Models

Conventional database management systems provide transactions as the atomic units of work. An activity that runs as an atomic transaction is guaranteed to satisfy the atomicity, consistency, isolation and durability (ACID) properties. However, executing a long running activity as a single transaction is not strictly necessary in most cases, and can significantly delay the execution of short transactions. For example, if purchase order processing is run as a single transaction, locks on the inventory records and the budget records may be held for a long time, severely limiting database concurrency. When these steps involve several distributed servers, commit processing is also expensive, and the transaction can run only when all servers are available simultaneously.

To address this problem, numerous advanced transaction models have been proposed [6]. These include *Sagas* [8], *nested transaction model* [13] and *flexible transaction model* [5]. But these models are not suitable for operating in real working

environments. They are too database-centric, which provides a nice theoretical framework, but limits the possibilities and flexibility of the models. Workflow models have, in general, richer semantics and are more apt to be used in commercial products. In fact, it has been proved that workflow models are a superset of advanced transaction models by implementing several transaction models using a single workflow system [2]. The unique features of workflow models are described in the next section.

2.2 Workflow Models

A *workflow model* is a directed acyclic graph in which nodes represent steps of execution and edges represent flow of control and data among the different steps. The building blocks of a workflow model are defined below :

- *Workflow* is a description of sequence of steps required to achieve a particular goal. It should have start and termination conditions and additional data for audit and control.

- *Task* defines a step within a workflow. Each Task has a name, a type, a set of pre and post conditions and scheduling constraints. A task requires a set of input and output parameters. There are two types of directed arcs that connect tasks, one is *data flow* arcs and another is *control flow* arcs.

- *Flow of Data* specified through data flow arcs between the tasks. A data flow arc maps the output parameter(s) of a task to input parameters of one or more tasks enabling them to exchange information.

- *Flow of Control* specified by control flow arcs between tasks, is the execution dependency between the tasks. A control flow arc is associated with a condition. This mechanism provides the functionality for defining branching, sequential/parallel execution and alternative execution steps.

- *Conditions* specify the circumstances under which certain events will happen. Conditions define labels of the control flow arcs and specify whether the control

flow should take place or not.

A *Workflow Management System* (WFMS) provides support for modeling, executing and monitoring the workflows. A WFMS considers four different sets of entities: users, tasks, programs and data. A WFMS automates the flow of control and data between tasks, and ensures the tasks are executed only by the valid users. Existing advanced transaction models solve only part of the problem. For instance, Sagas [8] provide a limited form of flow of control and data between tasks, but lack any reference to users or programs. The ConTract model [6] provides flow of control and data similar to a WFMS, but does not include users into the system.

The most important feature of WFMS is their ability to describe an organization and adapt the definition and execution of workflow processes to the particular characteristics of that organization [2]. In a WFMS, the organization is described in terms of the roles, hierarchical levels and persons associated with it. A person can assume several roles such as manager, programmer, assistant etc., and a role can be assigned to several persons. When a task is defined, the workflow designer must specify who is responsible for the execution of the task. This can be specified using a role, in which case all the persons that fit in that role are eligible to execute the task. This provides a great deal of flexibility in executing a workflow. Thus, tasks do not materialize automatically, as in advanced transaction models, but they are managed with direct user intervention. Even tasks corresponding to programs that do not require human input for execution are associated with users who can monitor their progress and are responsible for their execution. The user can stop a task, restart it, force it to finish, and so forth, independently of the rest of the workflow. This mapping between users and tasks is possible in WFMS because of the granularity of the tasks, which is that of applications, and much larger than that of traditional transactions.

Regular users interact with the system using *worklists*. A worklist is a list of workitems associated with the user. Each workitem is a task that belongs to the workflow being executed and that has been tagged to this user, and possibly also to others, for its completion. The same workitem may appear in several worklists simultaneously. However, as soon as a user selects a workitem for execution, it

disappears from all other worklists. This mechanism can also be effectively used to perform load balancing in the execution a workflow.

2.3 Example : A Bank Transaction Application

Let us see how a workflow system can be designed for a typical banking environment. We cover only a single type of banking transaction : cheque processing.

The following is the series of required operations :

- Scan the cheque (or other forms)
- Process the image to extract the values of various fields
- Verify the signature and the account balance
- Retrieve money and deliver it, or transfer money to the other account, depending on the type of transaction

Each of these steps is performed by one task. In addition, we require some auxiliary tasks, depicted in figure 1.

The activity starts when a customer submits a cheque to the system. It is assumed that all the pertinent information about the account holders is recorded in the database. This data is external to the workflow system.

The SCAN task controls a scanner, which reads the documents and produces image files. If multiple related documents are input, they are entered into a *folder*. Thus the relationship between the documents is maintained even though they are processed separately. This is the case with transfer of money from one account to another : one document is the cheque corresponding to the source account and attached with it is a pay-in slip for the target account. This step is depicted as a separate task in the diagram, though it can be combined with the SCAN task as per convenience.

The next step is to extract data from the image. Image processing techniques may be applied to identify the various fields in the document. A work object is created with data from these fields. Depending on the *key* or some such distinguishing

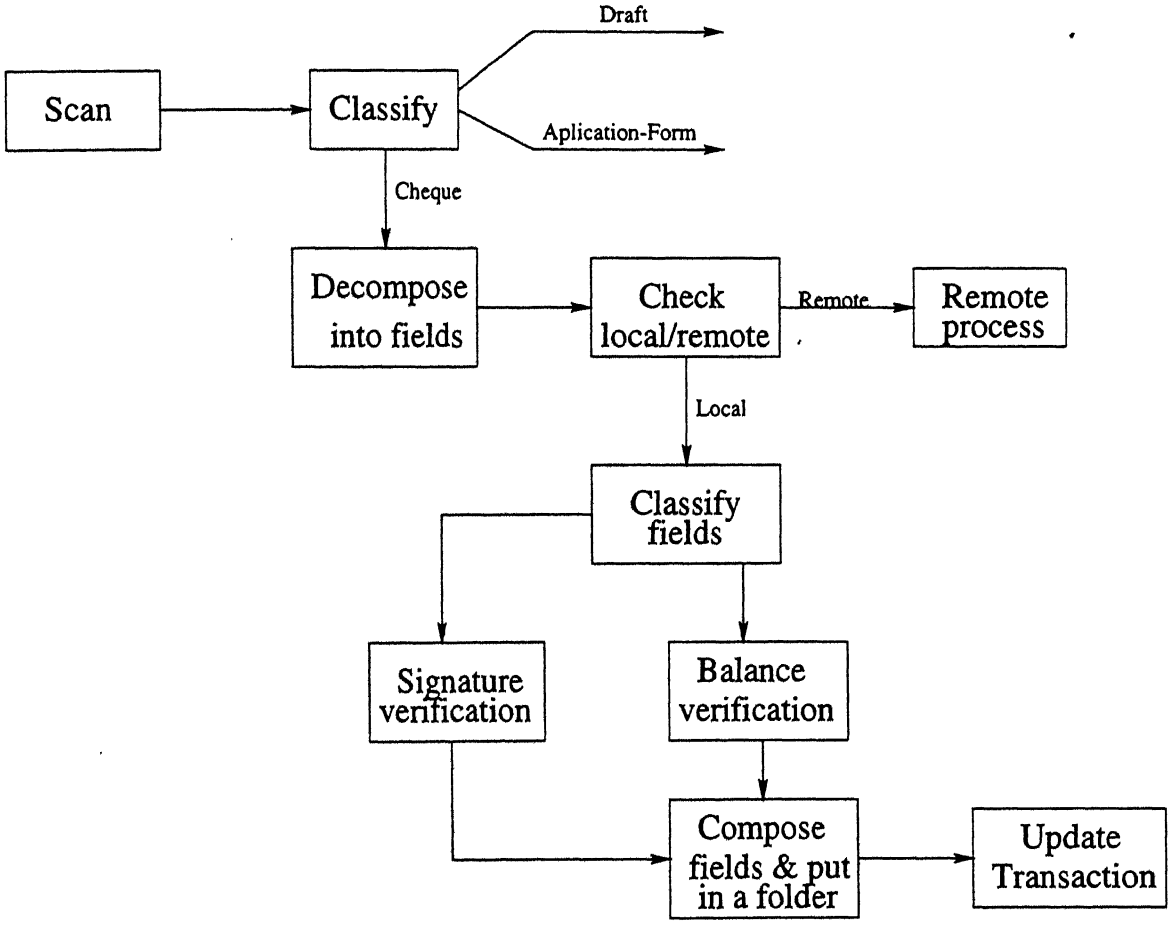


Figure 1: Workflow of a banking application

characteristic (in reality, this may even be the colour of the form), the type of the document (cheque, pay-in slip, loan request form, etc.) is identified. Accordingly, the object is routed to different tasks. For brevity, we further explain only the cheque processing module. The workflow for the other modules can be designed on similar lines.

A cheque contains fields such as the account number, signature of the account holder, value of the cheque, etc. In addition a *bank code* field identifies the bank for which the cheque is drawn. This is used to identify if the cheque requires local or remote processing.

Now, if the cheque corresponds to a remote bank, it has to be sent to that location and processed there. This may take a long time (even a few days) to complete. On the other hand, for local processing, we have to verify the signature and availability of the requested amount in the account. These two steps can proceed in parallel at two different sites. Sub-objects are created and routed to these two tasks. A pattern matching system can be used to compare the signature with that in the database. Or a user may manually run this task. The balance verification task may proceed non-interactively.

Both these tasks need to access the external database for customer accounts. These objects are transacted through a different path than the work objects (usually through direct SQL).

The sub-objects later combine at the COMPOSE task. If the transaction is a single cheque processing, it forwards it to the UPDATE task. If there is an associated pay-in slip, it task waits for the processed slip before forwarding to the UPDATE task. This is done by waiting for the detached folder components. The updating is done by crediting/debiting relevant accounts. The workflow is committed at this point.

In the chapters that follow, we show how this application can be implemented.

CHAPTER 3

System Architecture

In this chapter, we present a functional overview of a workflow model, outlining the modules. We first discuss the model proposed to be the standard for all workflow systems. Later, we briefly describe the workflow model proposed and implemented by us.

3.1 The *WfMC* Reference Model

To achieve interoperability between workflow systems, a standardized set of interfaces and data interchange formats is proposed by the **Workflow Management Coalition** [14]. The architecture of a workflow system may deviate from this, but it is acceptable as long as the interfaces between various modules conform to the standard. The benefit of such a standard is that different modules from different products can safely co-exist in an environment.

The reference model is depicted diagrammatically in figure 2. The various modules in the model are explained below :

1. **Process Definition Tools** : A variety of tools may be used to analyze, model, and describe a workflow. The interface of these tools with the rest of the system is termed the *Process definition import/export interface.* It provides a common interchange format for the following types of information :

   - Process start and termination conditions

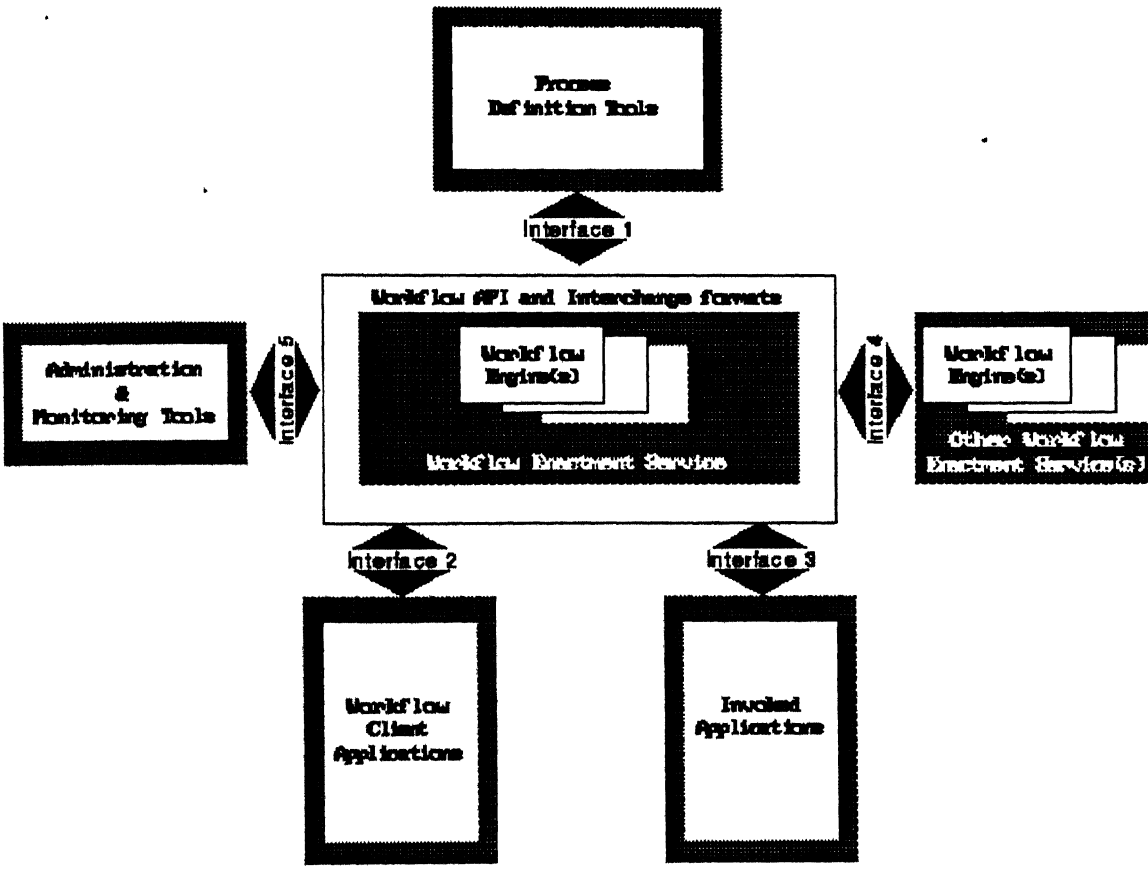


Figure 2: WfMC Reference Model

- Identification of activities within the process, including associated applications and workflow relevant data
- Identification of data types and access paths
- Definition of transition conditions and flow rules

2. **Workflow Enactment Service** : The workflow enactment service provides the run-time environment in which one or more workflow processes are executed. This may involve more than one actual workflow engine. The enactment service is distinct from the application and end-user tools which are used to process items of work. A wide range of industry standard or application specific tools can therefore be integrated with the workflow enactment service to provide a complete workflow management system.

3. **Workflow Client Applications** :

   The workflow client application is the software entity which presents the end user with his work items, and may invoke application tools which present to the user the task and the data relating to it, and allow the user to take actions on it. The workflow client application may be supplied as part of a workflow management system or may be a third party product (such as an Email product) or written specially for a given application. There is thus the need for a flexible means of communication between a workflow enactment service and the workflow client application which would provide a series of functions for connecting to the service and for obtaining and processing items of work.

4. **Invoked Applications :**

   These are the external applications (e.g., Email or other document managing programs) that are invoked as part of the task. The format of their interaction with the workflow should also be standardized.

As these standards are still evolving, they are not discussed here in detail.

3.2 The Proposed Workflow Model

The schematic diagram of the proposed workflow model is given in figure 3. Each task runs as a separate process. These are the *workflow client applications* mentioned in the reference model (section 3.1). The tasks are controlled and co-ordinated by a *Task Server*. The task server maintains the state of each task and schedules the tasks and events according to the specification. The workflow monitoring tool interacts with the task server to present the user with continuous updates of the activity in the workflow. The user can know how a work object is getting routed and its status.

We defined a language, called the *workflow language*, the primary objective of which is to define the work objects in the system. The language is a full-fledged programming language with data abstraction facilities. Objects can be defined with data and methods encapsulated. Another significant use of this language is for writing the code of the tasks. The language provides the `Task` abstraction and other built-in functions which are essential for it. The code written in this language is compiled into the Workflow Assembly Language (*WASM*).

Each task has associated with it, an interpreter for the Workflow Assembly Language (*WASM*). This will be used to execute methods on the objects as they come to this task. A task can invoke *external applications*, which may be full-fledged programs by themselves.

There are two more servers, which together with the task server, provide the *workflow enactment service* mentioned in the previous section. The object server provides the storage and retrieval facilities for the objects. It has lock management and failure recovery capabilities. It uses an RDBMS for its storage and provides an object-oriented interface to its clients. The object server also acts as an interface to the external databases. It provides storage and retrieval facilities for objects which are required by the tasks, but are different from the *work objects*.

The queue server manages the queues for various tasks. The task type acts as the queue identifier. There may be many instances of the same type of task, all of which share the same queue. The queues need to be stored in a permanent storage, because the objects may stay in the queue for a long time. The queue server utilizes the services of the object server to store and retrieve objects in the database.

The *workflow monitoring tool* can be used to visualize the execution of tasks and the flow of work objects. This tool also allows the user to query the workflow in a variety of ways. For example, one can find out the current state of a particular task or the location of a particular work object.

All the communication between the modules takes place by means of *remote procedure calls* (*RPC*).

The next two chapters discuss the detailed design and implementation of the task management and the task server. The object management and object server are discussed in [10]. For the sake of completeness, an overview of the object model is given in the next section.

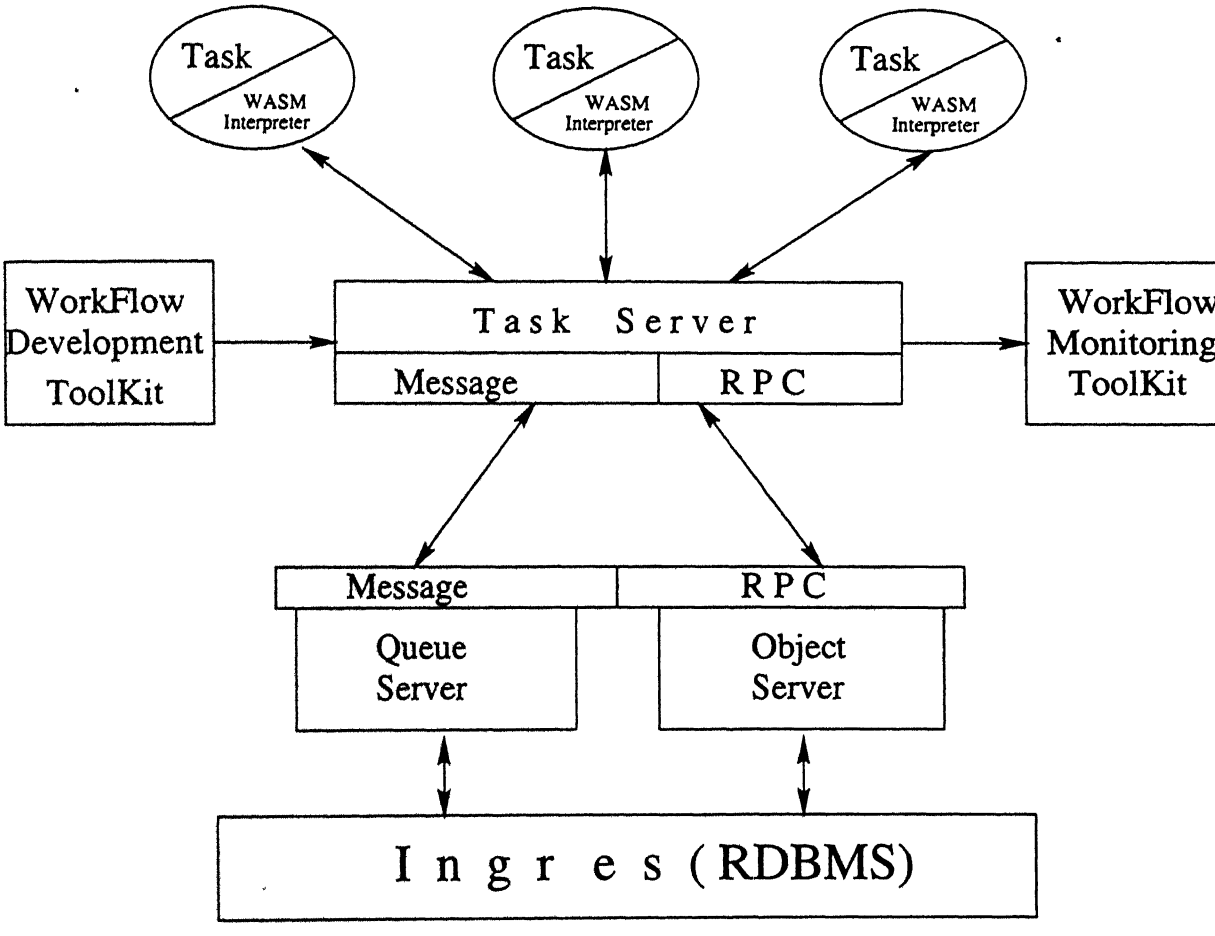


Figure 3: The Proposed Workflow Model

3.3 An Overview of the Object Model

The definition of work objects, their storage and retrieval, and the maintenance of queues are the main features covered in the object model.

3.3.1 The Workflow Language

The objects in the workflow are defined in the *workflow language* (or *Object Definition Language*). The main features of this language are mentioned below :

- The language has a set of standard scalar data types : **integer**, **float**, **string**, **boolean**, and a set of structured data types : **set** and **container**. The set and container are two important data structures that are very useful in workflow applications. Containers are used to maintain relation between its member objects, though the objects maintain independent existence.
- The `class` abstraction is useful for defining objects. Attributes and methods can be encapsulated into a class definition.
- The `Task` abstraction is used for defining code for a task, described in more detail in section 4.4.
- Built-in functions are provided for invoking external applications. The API functions used by a task for interaction with the task server are provided as built-in functions in the language.

A compiler for this language is developed. It translates the method code and task code into the Workflow Assembly Language. This architecture-independent code can be retrieved by any task and interpreted on any machine.

3.3.2 The Object Server

The object server takes care of the storage and retrieval of objects to and from a relational database. In addition to objects, the *schema* i.e., the class definition of objects and the compiled code for the methods is also stored. The object server uses

Ingres at its back end and represents the objects and schema as relational tuples. It ensures consistent access to the objects by means of *locking.*

The **Queue Server** is the server module which closely interacts with the object server to maintain the queues for various tasks. When a task requests, it takes the next object from the queue corresponding to the task and returns it; similarly stores the output of the task in the appropriate queue depending on the routing.

CHAPTER 4

The Task Model

The task model defines the application task management in a workflow. Application tasks are defined, maintained, scheduled and traced by the task model of a workflow. It is concerned with the specification of the task as a state transition system, its execution through the states by maintaining consistency and keeping track of the activities within a task for every job. Accordingly, the following modules can be identified :

- Task Specification
- Task Server
- Task API

First, we expand on the task model concepts presented in the previous chapters and then explain each of these modules in detail.

4.1 Task Definition

A task is defined as a state transition system. A job in a task starts at an initial state and terminates in one of the final states, after passing through a sequence of intermediate states. A state can be *internal* or *exposed*. The task server is aware of only the exposed states i.e., it is informed only when a task changes to an exposed

state. Transitions to internal states will be hidden from the task server. This implies that one can query only about the exposed states of a task. Inter-task interaction is possible only through these states.

The state diagram acts as a guideline for the execution of a task. The code to implement this state diagram is written separately. The advantage of clearly defining the states in the execution of the task is that an external module can find out what the task is doing by looking at its current state. It also helps in locating a given work object more precisely – at the state level. In addition, the state diagram of a task can be analyzed by data flow analysis techniques to detect potential flaws in the execution plan. One such problem is a task not terminating in any of the pre-defined states. State changes for each task are logged at the task server, along with the work object ID, as described in section 5.3.

Each state is associated with two pieces of code - ASSERT code and LEAVE code. The ASSERT code is executed when the state is *asserted* (i.e., entered). Similarly, the LEAVE code is executed when the task leaves the state and goes to a new state.

There are a set of pre-defined states – FORWARD, COMMIT, ABORT and EXCEPTION. A task finally reaches one of these states to mark the end of its execution (for the current work object). The work object will then be routed according to the routing specification. Typically, it will go to the next task in the pipeline. These pre-defined states are described below :

- FORWARD The task is done with the current work object and has committed successfully. The next work object for this task is obtained from the queue.

- COMMIT This state is asserted when the whole transaction has committed successfully. The work object is no longer in existence after this.

- ABORT Abort the current transaction. Failure recovery by compensation will take place as described later.

- EXCEPTION When this state is asserted, an *exception* is said to be raised. This is to deal with any abnormalities in the task. For instance, a task may raise an exception if it detects some system problem. An exception processing task then takes over the current work object and decides what to do. If the problem

is with the work object, it may recover the object and send it back to this task for processing again.

To illustrate, consider a task which does the document scanning part of a bank application. It first scans the image (in the state SCANNING), does some image processing to read the data from the document (state PROCESS), stores it in the database (state STORE), then forwards the created object to the next task. In case the scanning operation fails, or it finds some error while processing the image, it goes to the ABORT state. This is depicted in the following state diagram.

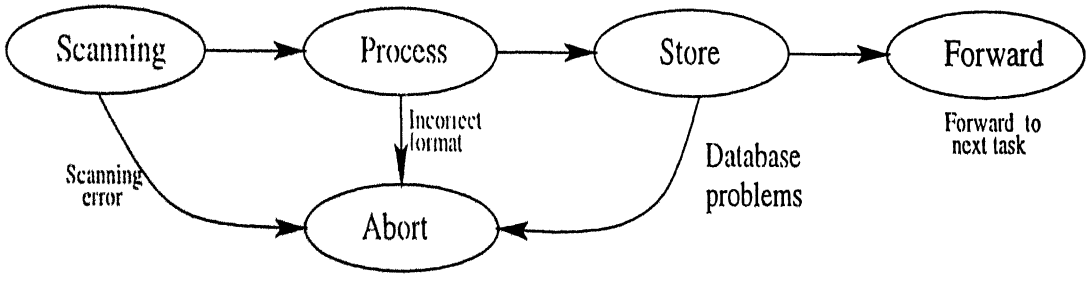


Figure 4: State diagram for the SCAN task

4.1.1 Events

Events may be raised for internal state transition in a task as well as for inter-task interaction. Accordingly, there are two types of events, *intra-task events* and *inter-task events*. An intra-task event is simply an event which labels a state-transition edge. If an event E is a label from state S_1 to state S_2, then an event E may be raised when the task is in state S_1. On this event the task leaves state S_1 and asserts state S_2.

An inter-task event is more complex and is defined using the temporal E-C-A model of event trigger specification. An E-C-A event is written as *on event E if condition C holds then execute action A*. In a temporal model there may be temporal

constraints between E-C and C-A, called ECTime and CATime. A temporal constraint is a temporal condition which evaluates to a temporal interval. The temporal operators used are

BEFORE *time*
AFTER *time*
AT *time*

which may be composed using temporal compositions, AND/OR (to imply intersection/union of intervals). In addition, the time may be qualified with RELATIVE or ABSOLUTE, to indicate whether the temporal interval is to be computed relative to the present time or the absolute time.

An exposed state specifies E-C-A for an event using the clause ON EVENT, to say what to do when this event is raised and the task is in the corresponding state. The respective state is the state when the task may honour the event.

An inter-task event from one task to another may send certain parameters, which may be used in the condition evaluation or action execution.

An inter-task event may be synchronous or asynchronous. A synchronous event does not return after raise until the action is executed or there is a timeout of the temporal intervals. An asynchronous event, on the other hand, returns immediately after the event is registered by the task server.

An inter-task event may also send a work object as a parameter, which is uncommitted.

For example, an E-C-A triplet for representing an event that an updating task would receive from an account verification task might be specified as follows :

```
ON EVENT ev_update (t1, t2)
        AFTER 1 HRS AND BEFORE 2 HRS RELATIVE
CONDITION ''...''
        BEFORE 10 MINS RELATIVE
ACTION ''...''
```

Here, t1 and t2 are the event parameters. They can be used in the condition and the action code; their values can be passed at the time of raising the event.

Control and data flow dependencies can be enforced by means of events. Suppose a dependency is such that a task T_i can enter state S_i only after task T_j enters state S_j. This can be achieved as follows : T_i, before entering S_i, waits for an event from S_j; T_j will raise the event once it enters S_j. Now, T_i will wakeup and proceed.

The task API has calls to raise an event, wait for an event and others. See section 4.3 for the details.

Global Events: Some events can be specified to be global i.e., they are not related to any task. These can be used for maintenance jobs such as generating statistics or controlling the environment. These can be specified to be periodic. The scheduler executes the actions corresponding to these events when the time expires.

Global events may be raised by any task. In addition, all global events are raised once at the time of system startup. For each event, when the time comes, the action is executed. As an application of this feature, we can arrange for documents to be faxed in the night hours by scheduling the FAX task appropriately. All work objects (documents) coming for faxing wait in the queue till the task is spawned. To implement this, we can have a global event, FAX_EVENT, which as part of its action code, invokes the FAX task. The E-C-A can be specified as follows :

```
ON EVENT FAX_EVENT GLOBAL
CONDITION "true"          /* always execute the action */
        AT 21:00:00 ABSOLUTE
ACTION "exec('fax.exe');"
```

In this way, we can cause a task to be scheduled at a specific time.

4.1.2 Routing Specification

The *routing specification* forms an important part of a task definition. Here, we declare what is to be done with the objects after getting processed by the task. Most commonly, this involves specifying the next task to which the objects will go,

in each of the four possible termination conditions - COMMIT, ABORT, FORWARD and EXCEPTION. In each case, we can give a short sequence of statements to be executed before the object is sent out. This is called the *routing script*. The code decides the destination of the work object, possibly by examining the work object.

Our bank application may use this feature to route the cheque based on its value (i.e., the amount for which it is drawn). It may be required that cheques of higher amount should pass through additional tests or should be processed only by tasks in a certain role. The routing script for this is written as follows :

```
ON FORWARD
        if (obj.amount > threshold)
                route(PRIVILEGED_TASK);
        else
                route(NEXT_TASK);

ON EXCEPTION
        route(EXCEPTION_TASK);
```

The variable `obj` in this part of the code refers to the current work object of the task.

4.1.3 Task Invocation

A task may be invoked in a number of ways.

- **Manual Invocation** : This is the ordinary method of invocation where a user starts up a task. The process of logging in takes place at this point. The name of the user is obtained from the process environment, or read from the terminal (in the latter case, the password is also read in).

- **Auto Invocation** : When work objects arrive for a task that is not already running, the task server can invoke the task automatically. This is called *autoloading* of the task. Tasks which are needed infrequently and execute only

for a short amount of time are usually autoloaded. These tasks usually do not require any interaction with the user. Such tasks may exit after processing a single job.

- **Periodic Invocation** :

  Some tasks are needed to be run periodically. These are most commonly the supervisory tasks run by the administrator. For instance, an administrative task may check for any abnormalities in the system every few hours. Or, a system may use such a task to send reports on its transactions twice a day to another location.

- **Compensation** : A task may be run in compensation for another task. When a task aborts, the task server invokes its compensating task, if one is specified for the task. This concept is described in more detail in the next subsection.

4.1.4 Task Compensation

The idea of task compensation comes from [8]. A *compensating transaction* is a transaction with the opposite effect of an already committed transaction. It is intended to undo the visible effects of a previously committed transaction. Thus, for a task which deducts some amount from an account, the compensating task will add the same amount to bring it to the previous state.

In a workflow, a task can define its compensating task. If a task aborts, then the compensating tasks for all the previous tasks (in this workflow) have to be executed in the reverse order. The task server looks at the log file to find out the tasks that are already committed. For each such task, it then invokes the compensation task.

4.2 Task Specification

All the information mentioned in the previous sections can be specified to the system by means of a **Task Specification Language**. The task server reads a specification file written in this language and stores it in its own internal database, called the *TaskBase*.

The specification is a sequence of task definitions, where for each task, we give the following :

- Task Attributes
  - **Task type**
  - **Users** : who are eligible to run this task
  - **Roles** : the roles in which the task should be executed
  - **Invocation** : type of invocation, as described in the previous section.
  - **Compensated By** : type of the task that acts as the compensation task when this task fails.
  - **Compensation To** : type of the task for which this is being executed as compensation.
  - **Task Code** : name of the file containing the executable code for this task.
- A sequence of state definitions, where
  for each state, we specify
  - whether it is **internal/exposed**
  - ASSERT code : The code that is to be executed when the task enters this state.
  - LEAVE code : The code that is to be executed when the task goes out of this state.
  - ECA triplet for each event accepted in this state
- Routing specification

The complete grammar of the specification language in BNF Notation is given in Appendix A.

The user authentication in this model is very simple. A task specifies a list of roles in which this task must be executed. A more complex scheme would be a *hierarchy* of roles, which can possibly correspond to the organizational hierarchy.

The routing and the event specifications together can be termed *inter-task description*, as it describes the interaction between two tasks at a higher level.

A graphical front-end is developed for generating the task specification easily. One can draw the state diagram and enter the data using an easy-to-use graphical interface; the specification file is automatically generated. This tool is described in chapter 5 of [10].

4.3 Task API

The **API** is a set of function calls, which a task will use to interact with the rest of the workflow system. All of these calls are implemented as remote procedure calls by the task server.

Typically, the code for a task is written in our workflow language, in which these are provided as built-in functions. One can also write the task code in an external language (like C or C++) and still use the API. It does this by linking the API library with its own code. Hence it becomes significant that the semantics of the API be well defined. External applications conforming to this standard can be easily used, thereby achieving interoperability.

The following functions constitute the API :

1. `init(TaskType, User, Password, Role)`

   This is the initialization function for every task. A user can *login* to the system by giving his name and password. He also declares the role in which he wants to work.

   First, the task server is located from the information given in a *configuration file*. It contains the name of the machine where the server is running and the port number on which it will receive messages from the tasks. Then, a message is sent to the task server asking for registration of the task. The user is authenticated at this point. It checks whether the user (in that role) is allowed to run this task. If everything goes well, the task is registered and a **Task Identifier** is returned, which identifies this particular instance of the task. Multiple instances of the same task may be in execution at the same

time. Since they all do the same kind of processing, the work object queue is shared among all of them.

2. `assert(stateID)`
   A transition to a new state is effected by this call. If the transition is valid from the current state, the leave code for the current state and then the assert code for the new state are executed. If the new state is an exposed one, the task server is informed of the state change.

3. `GetCurrentState()`
   This is an enquiry function which returns the current state of the calling task.

4. `raise(EventName)`
   Raise an internal event. The effect is to change the state to the target node of the edge that is labelled with this event name.

5. `raise(TaskType, Users, Roles, WorkObject, EventName, EventType, EventParameters)`

   This is the function to raise an inter-task event. The target task type is identified by its type and the possible users running it (and their roles). The task server finds the task which matches these parameters (It selects one arbitrarily if there is more than one matching task). The type of the event raising is either SYNC/ASYNC, indicating a synchronous event or an asynchronous event respectively. The values for the event parameters are given as the rest of the arguments. The work object is passed uncommitted to the target task. Once an object is passed like this, it is no longer available at this task and hence should not be used any further to this call. Run-time checks are inserted into the code to ensure this. Any violation is caught and an exception is raised.

   All the parameters are packed into a request structure and passed to the task server. The task then waits for acknowledgment if the event is synchronous. The processing of the event is done at the task server as described in the previous section.

6. `WaitOnEvent(event, timeout)`
   Keeps the task waiting until the specified event is raised for this task, or timeout occurs. The return value identifies what happened. If timeout is 0, it is a blocking call.

7. `WaitOnEventEnd(event, timeout)`
   This call is similar to the above, except that the task waits until the action for the event is executed.

8. `down()`
   This is called before exiting. The task server notes that the task has gone down and updates its tables accordingly.

In addition to these task-related functions, the API also has two functions for invoking external applications.

- `exec(command)`
  The command given as the argument is executed as another process. The task waits till the end of the invoked process.

- `launch(command)`
  This function is the same as the previous one, except that the task does not wait for the end of the application. The new process proceeds independently of the task.

All these functions return an error value (-1) on failure.

The execution plan of a task defines how the task execution goes for different work objects. The protocol for a proper execution plan is as follows :

- Register with the task server by calling `init()`
- For every work object, do the following :
  - Assert the initial state
  - At each state, do the required processing and go to the next state by calling `assert()`, or by raising an internal event.

- Assert one of the pre-defined final states after the work object is processed completely.
- Raise inter-task events from any of the intermediate states, as needed, by means of `raise()`.

- Finally, shutdown by calling `down()`.

If a task is not initialized properly, all the subsequent task API calls will fail. In case a task aborts abruptly, without a proper shutdown, the task trace remains in the task server, and the task administration will remove the task from the task server.

As can be observed easily, there is a significant commonality among all tasks in the sequence of calls that they make. For tasks that are written in the workflow language, the code for initialization and looping over the work object is generated implicitly. All that a task writer needs to write is the actual processing part. The code for initialization and looping over the work object is generated implicitly. To do this conveniently, a new abstraction, `Task` is defined in the language. This is explained in the next section.

4.4 The Task Abstraction

The `Task` is a construct in the workflow language useful for writing tasks, which need to obey the above mentioned protocol. This construct is very similar to the `class` construct. The language constructs are explained in the next chapter; the `Task` construct is described here for convenience.

```
Task name {
    internal objects;
    methods;
    main() {
        ⋮
    }
};
```

The startup function should be named `main()`. Since typically, a task is run on a single machine with a user attentive to it, the task presents itself with a good user interface. All input is done through dialog boxes and all output goes to another window on the screen. This is done by redefining the standard I/O functions described earlier.

The interpreter does the following when such a task is spawned :

1. It gets the details such as the user name, password and role by the *login process* and calls `init()` to initialize the task.

2. It implicitly generates a loop to do the following :

   - call `GetWorkObject()` to get the next work object from the queue
   - call the `main()` function of the task.

   The `main()` function can access the work object by using the pre-defined global variable `workobject`. It may modify this variable by calling methods on it. When the function terminates, this object will be routed as described earlier.

3. If all the objects are exhausted, it calls `down()` to shutdown the task.

Thus, the main function appears as if it is written for a single object. The fact is that it is called once for each object.

4.5 Implementation Details

Ensemble Processing

In the previous sections, the processing of a task is described in terms of individual work objects. A task gets a work object from its queue, processes it and sends it to another queue as determined by the routing script. But in reality, groups of objects, called *ensembles* are exchanged between the tasks and queues. An ensemble is a package of objects of similar type. The main advantage of this is *efficiency.* It is very efficient in terms of the utilization of network resources, to send and receive

groups of objects instead of a single object. The number of requests from the task will be reduced.

The processing of work objects now proceeds as follows : When a task first calls `GetWorkObject()`, a whole ensemble of objects is retrieved. The task then processes objects from this ensemble. When all of them are processed, the objects in the ensemble are sent out, routed according to the routing script and the ensemble is refilled with new objects.

CHAPTER 5

The Task Server

The task server is the heart of a workflow management system. As its name implies, it is the process which manages all the tasks in the workflow. The task server keeps track of what the individual tasks are doing. It maintains their current state, current work object and others. Another very important job that the task server does is *scheduling.* It keeps track of the events that the tasks want to exchange among them; it evaluates the condition and action associated with the events and delivers them to the appropriate task. Finally, the task server maintains logs at each appropriate point in the workflow. This helps in workflow monitoring and error recovery.

5.1 Scheduling

Scheduling is the problem of processing workflows by submitting various tasks for execution, monitoring various events, and evaluating conditions related to inter-task dependencies [11].

The scheduler part of the task server is very crucial to the system. There are many approaches to scheduling in the context of multi-database transactions and workflows. Some schedulers are based upon predicate petri nets, finite state automata and propositional logic. Our model is closer to the *event-condition-action* rule interpreter of [3]. An overview of the scheduling strategies is given in [11].

5.1.1 Event Scheduling

For each event registered with the scheduler, it has to check the condition at the specified time and if it evaluates to true, execute the corresponding action. The time delay between the condition testing and action execution will also be given in the E-C-A specification.

The algorithm of the scheduler can be described as follows :

Events raised by various tasks are maintained in an *Event Queue*. There are two types of events in the queue – those for which *condition* has been evaluated (eventQA) and those for which both condition and action are pending (eventQC).

```
Take event from eventQueue;
Look for the receiver task;
If (the receiver task is waiting for this event)
  send an event signal to it and remove it from the wait;
Push the event for Condition evaluation to eventQC;
Write to log the event name and current time;
Take an event from eventQC;
if (the time is reached)
 {
   evaluate Condition;
   write to log the event name, current time and condition result.
   if true then schedule it to eventQA;
   else if (it is still temporally relevant)
        push it back to eventQC;
 }
Take an event from eventQA;
if (the time is reached)
 {
   execute A;
   if (some Task is waiting for end)
        send end signal to the task and remove
        it from waiting;
```

```
    if (the event is global and to be re-spawned)
        then reschedule the event to eventQueue;
}
```

When the task server receives the indication of an event from a task, it is pushed into `eventQueue`, and the intended receiver task of the event is noted. Periodically, the scheduler polls this queue and puts them into `eventQC` for condition evaluation. If any task is waiting for the occurrence of the event (through the `WaitOnEvent()` API call), it is sent a signal. When the condition time expires, the condition code will be evaluated by the workflow language interpreter. If the condition is satisfied, the event will now be moved from `eventQC` and entered into `eventQA`. The action is executed at the appropriate time. If a task is waiting for the end of the event (through `WaitOnEventEnd`), it is signalled. Periodic global events are entered into the `eventQueue` again.

5.1.2 Implementation Details

The task server has to service the task API in parallel to the event scheduling. It receives the API requests as messages over a *socket*. There are usually a number of tasks that have open connections with the task server. The task server polls these connections to receive messages. It does this by means of the `select()` system call. Periodically, it breaks out of the polling loop (by specifying the *timeout* parameter to `select()`) and services the events as described in the algorithm above.

The task servers maintains an *active task list* which stores information about the currently executing tasks. For each task, it remembers its current state and the work object it is processing. When a task asks for registration, it verifies the user and role for authentication. If it succeeds, the task is noted as registered and entered into the active task list. It then receives messages when exposed states are asserted by the active tasks. Finally, it removes the task from the list when it calls `down()`.

5.2 Interaction with the Monitoring Tool

The task server closely interacts with the monitoring tool to keep the user updated about the state of the workflow. Periodically, it informs the monitoring tool of the changes that have taken place since the last update. The tool will then update its display to depict this state. The details of monitoring tool are explained in section chapter 6 of [10].

5.3 Logging

The task server maintains a log of activities in the workflow. Logging is useful in a number of ways :

1. **Error tracking** : When a task fails (exits abnormally), we can immediately know what its state of execution was, before exiting. We can also find its state and the work object it was processing. So we can undo whatever the task has done in the failed transaction.

2. **Querying the workflow** : A variety of queries can be made about the workflow. Typical examples are :

   - How much time did a task T spend in state S?
   - How much time did a work object w stay with a task T?
   - What are the tasks through which a work object w has passed since its creation?

All the information is not stored in a single log file. Instead, two logs are maintained - one for the work objects and one for the tasks.

The format of the records in the work object log is as follows :

`Work Object` *ID* `entered task` *ID* `(type` *type*`)` `at` *date, time*

The format of the task log records is as follows :

Task *ID* (type *type*) entered state *name* with work object *ID* at *date, time*

Task *ID* (type *type*) raised event *name* to task *ID*

Event *name* condition for task *ID* result : *result*

Event *name* action for task *ID* result : *result*

These two logs together store the following information :

- Asserting and leaving times of a state along with the work object.
- When a task gets a new work object, the time and the work object ID are logged.
- Event raising times - the time when a task raises an event.
- Event processing times - The time and result of the condition and action associated with each event.

CHAPTER 6

Conclusions

In this thesis, we discussed the design and implementation of a workflow model. First, we introduced the concept of workflow and its relevance in the field of database transactions. Workflow models are compared to advanced transaction models with respect to their features and implementability. The task model and the task server parts of the workflow are considered in detail. The rest of the work is done as a separate thesis [10]. An example application of workflows is described throughout the thesis.

We implemented a prototype of the workflow model presented in this thesis. The implementation has been carried out with the help of a number of programming languages and tools. A major portion of the code has been written in C++. The UNIX tools *lex* and *yacc* are used for parsing the specification files. A full-fledged compiler has been developed for the workflow language. **Tcl/Tk** has been used for the graphical user interface programming. The UNIX mechanisms of RPC and message passing through sockets are used for communication between processes on different machines.

6.1 Future Work

Extensions to this model and implementation are discussed below :

The problems of failure recovery and exception handling have been attempted here to a limited extent. Research is still going on to find general solutions to these problems [12]. A persistent messaging system in conjunction with a stable storage mechanism can be used to recover from failure within a task and failures due to communication. In addition to this, failures of DBMSs have to be taken care of.

Another requirement of our implementation is that the tasks should be connected to the task server all the time. If this constraint is relaxed, tasks can be made *mobile*. This makes it possible for a user to interact with the workflow from different sites at different times.

In our system, only the workflow specification and the task specification can be generated visually. Graphical user interfaces can be used in the specification of objects also. It should be possible for the user to define the object attributes by selecting the data types from menus. Ideally, code for the methods of objects and tasks could also be generated visually. The generation of programs visually (or *visual programming*) is a separate research problem by itself.

A number of research perspectives in workflow systems with specific reference to the **Exotica** product have been mentioned in [12].

Bibliography

[1] ACTION TECHNOLOGIES. http://www.actiontech.com/.

[2] ALONSO, G., AGARWAL, D., EL ABBADI, A., KAMATH, M., GUNTHOR, R., AND MOHAN, C. Advanced transaction models in workflow contexts. Tech. rep., IBM Almaden Research Center.

[3] DAYAL, U., HSU, M., AND LADIN, R. Organizing long-running activities with triggers and transaction. In *ACM SIGMOD Conference on Management of Data* (1990).

[4] DAYAL, U., HSU, M., AND LADIN, R. A transaction model for long-running activities. In *Proceedings of the 17th International Conference on Very Large Databases* (1991).

[5] ELMAGARMID, A., LEU, Y., LITWIN, W., AND RUSINKIEWICZ, M. A multidatabase transaction model for interbase. In *Proceedings of the 16th International Conference on Very Large Databases* (August 1990).

[6] ELMAGARMID, A. K., Ed. *Database Transaction Models for Advanced Applications.* Morgan-Kaufmann, 1992.

[7] FILENET TECHNOLOGIES. http://www.filenet.com/.

[8] GARCIA-MOLINA, H., AND SALEM, K. Sagas. In *ACM SIGMOD Conference on Management of Data* (May 1987), pp. 249–259.

[9] IBM ALMADEN RESEARCH CENTER. http://www.almaden.ibm.com/almaden/.

[10] KALADHAR, M. Design and implementation of a workflow model, part II : Object model. Master's thesis, Dept. of CSE, Indian Institute of Technology, Kanpur, January 1997.

[11] KIM, W., Ed. *Modern Database Systems.* Addison Wesley Publishing Co., Reading, Mass., 1995.

[12] MOHAN, C., ALONSO, G., GUNTHOR, R., KAMATH, M., AND REINWALD, B. An overview of the exotica research project on workflow management systems. Tech. rep., IBM Almaden Research Center.

[13] MOSS, J. Nested transactions : An approach to reliable distributed computing. Tech. Rep. MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[14] THE WORKFLOW MANAGEMENT COALITION. http://www.aiai.ed.ac.uk/wfmc/.

Appendix A

Task Specification Grammar

```
1   /*
2    *      Grammar Specification for the task description file
3    */
4
5   %union {
6           int     Integer;
7           char    *String;
8           char    *IDList[50];
9           int     NUMList[50];
10          Time    *Ptime;
11          Interval        *Pinterval;
12  };
13
14  %token ID NUM STRING
15  %token TASK TYPE USERS ROLES Initial FINAL STATE
16  %token EXPOSED INTERNAL
17  %token TOK_SYNC TOK_ASYNC
18  %token ON ASSERT LEAVING EVENT CONDITION ACTION
19  %token RELATIVE AND OR AT BEFORE AFTER WITHIN HRS MINS SECS
20
```

```
21  %token  ON ROUTE
22  %token  COMMIT ABORT FORWARD
23
24  %type <Integer> NUM state_type ev_type
25  %type <String>  ID STRING
26  %type <String> code
27  %type <String> const_time
28  %type <Integer> hrs_spec min_spec sec_spec
29  %type <Ptime>   time_value
30  %type <Pinterval>       time_spec simple_time_spec
31  %type <IDList>  id_list evparam_list opt_evparam_list
32  %type <NUMList> num_list
33
34  %type   <Integer>       condition
35  %type   <String>                route_stmt
36
37  %%
38
39  spec    : task_list             /* A list of task specs */
40          ;
41
42  task_list       : task_list task_spec
43                  | /* Nothing */
44                  ;
45
46  task_spec       :  type_spec User_list Role_list
47                     invocation_spec
48                     compensation_spec
49                     code_spec
50                     initial_spec final_spec state_list
51                     routing_spec
```

```
52                      ;
53
54  type_spec           : TASK TYPE ':' ID
55                      ;
56
57  User_list           : USERS ':' id_list
58                      ;
59
60  Role_list           : ROLES ':' id_list
61                      ;
62
63  invocation_spec     : INVOCATION ':' invocation_type
64                      ;
65
66  invocation_type     : MANUAL
67                      | AUTO
68                      | COMPENSATION
69                      | PERIODIC NUM
70                      ;
71
72  compensation_spec:  COMPENSATED BY ID
73                      |  COMPENSATION TO ID
74                      ;
75
76  code_spec           : TASK CODE ':' STRING
77                      ;
78
79  initial_spec        : Initial NUM
80                      ;
81
82  final_spec          : FINAL num_list
```

```
83                  ;
84
85  state_list      : state_list state_spec
86                  | state_spec
87                  ;
88
89  state_spec      : state_header assert_spec leaving_spec eca_list
90                  ;
91
92  state_header    : STATE NUM state_type
93                  ;
94
95  state_type      : EXPOSED
96                  | INTERNAL
97                  ;
98
99  assert_spec     : ON ASSERT code
100                 ;
101
102 leaving_spec    : ON LEAVING code
103                 ;
104
105 eca_list        : eca_list eca_spec
106                 | /* Nothing */
107                 ;
108
109 eca_spec        : ON EVENT ID ev_type opt_global opt_evparam_list
110                         time_spec
111                   CONDITION code
112                         time_spec
113                   ACTION code
```

```
114                     ;
115
116   ev_type           : TOK_SYNC
117                     | TOK_ASYNC
118                     ;
119
120   opt_global        : GLOBAL
121                     | /* Nothing */
122                     ;
123
124   opt_evparam_list: '(' evparam_list ')'
125                     | /* Nothing */
126                     ;
127
128   evparam_list      :  id_list      /* formal parameters */
129                     ;
130
131   time_spec         : time_spec AND time_spec
132                     | time_spec OR time_spec
133                     | simple_time_spec
134                     ;
135
136   simple_time_spec: AT time_value
137                     | BEFORE time_value
138                     | AFTER time_value
139                     | WITHIN time_value
140                     ;
141
142   time_value        : const_time
143                     | const_time RELATIVE
144
```

```
145                     | ID    /* evparam */
146                     | ID RELATIVE
147                     ;
148
149   const_time        : hrs_spec min_spec  sec_spec
150                     | NUM ':' NUM ':' NUM
151                     ;
152
153   hrs_spec          : NUM HRS
154                     | /* Nothing */
155                     ;
156
157   min_spec          : NUM MINS
158                     | /* Nothing */
159                     ;
160
161   sec_spec          : NUM SECS
162                     | /* Nothing */
163                     ;
164
165   id_list           :  id_list ',' ID
166                     | ID
167                     ;
168
169   num_list          : NUM
170                     | num_list NUM
171                     ;
172
173   code              : STRING
174                     ;
175
```

```
176 routing_spec     : /* nothing */
177                  | one_spec routing_spec
178                  ;
179
180 one_spec        : ON condition
181                    code      /* routing statements */
182                  ;
183
184 condition        : COMMIT
185                  | ABORT
186                  | FORWARD
187                  ;
188
189 %%
```